Algorithm:

- Step 1: Create a unique, unenterable minus state and a unique, unleaveable plus state.
- Step 2: One by one, in any order, bypass and eliminate all the non-minus or non-plus states in the TG. A state is bypassed by connecting each incoming edge with each outgoing edge. The label of each resultant edge is the concatenation of the label on the incoming edge with the label on the loop edge (if there is one) and the label on the outgoing edge.
- Step 3: When two states are joined by more than one edge going in the same direction, unify them by adding their labels.
- Step 4: Finally, when all that is left is one edge from - to +, the label on that edge is a regular expression that generates the same language as was recognized by the original TG.

Proof of Part 3: Converting Regular Expressions into FAs

- We prove this part by **recursive definition** and **constructive algorithm** at the same time.
  - We know that every regular expression can be built up from the letters of the alphabet $\sum$ and Λ by repeated application of certain rules: (i) addition, (ii) concatenation, and (iii) closure.
  - We will show that as we are building up a regular expression, we could at the same time building up an FA that accepts the same language.

Kleene's Theorem and NFA:

- Before we proceed, let's have a quick review of the **formal definition of regular expressions**.
- The set of **regular expressions** is defined by the following rules:
- Rule 1: Every letter of the alphabet $\sum$ can be made into a regular expression by writing it in **boldface: Λ** itself is a regular expression.
- Rule 2: If r_1 and r_2 are regular expressions, then so are:
  (i) (r_1)
  (ii) r_1r_2
  (iii) $r_1 + r_2$
  (iv) r_1
- Rule 3: Nothing else is a regular expression.

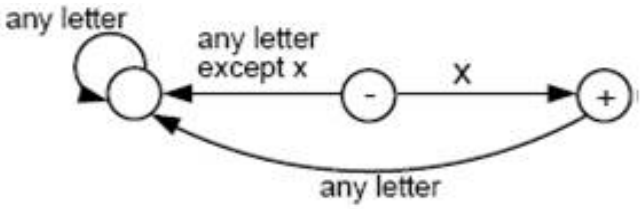
We now present proof of part 3 recursively.

Rule 1:

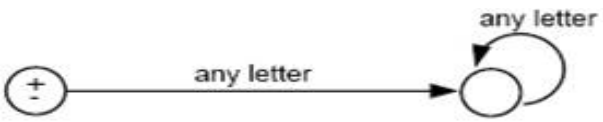
- **There is an FA that accepts any particular letter of the alphabet.**
- **There is an FA that accepts only the word Λ.**

Proof of rule 1:

- If letter x is in ∑, then the following FA accepts only the word x.



- The following FA accepts only :



Rule 2:

- **If there is an FA called** FA_1 **that accepts the language defined by the regular expression** r_1**, and there is an FA called** FA_2 **that accepts the language defined by the regular expression** r_2**, then there is an FA that we shall call** FA_3 **that accepts the language defined by the regular expression** $(r_1 + r_2)$**.**

Proof of Rule 2:

- We shall show that FA_3 exists by presenting an algorithm showing how to construct FA_3.

Algorithm:

– Starting with two machines, FA_1 with states $x_1; x_2; x_3; \ldots$, and FA_2 with states $y_1; y_2; y_3; \ldots$, we construct a new machine FA_3 with states $z_1; z_2; z_3; \ldots$ where each z_i is of the form $x_{something}$ or $y_{something}$.

– The combination state x_{start} or y_{start} is the start state of the new machine FA_3.

– If either the *x* part or the *y* part is a final state, then the corresponding *z* is a final state.

– To go from one state *z* to another by reading a letter from the input string, we observe what happens to the *x* part and what happens to the *y* part and go to the new state *z* accordingly. We could write this as a formula:

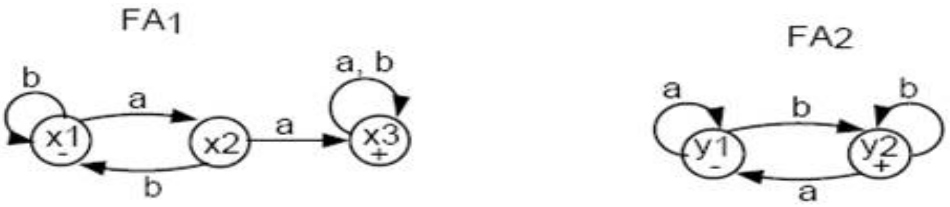
z_{new} after reading letter p = (x_{new} after reading letter p on FA_1) or (y_{new} after reading letter p on FA_2)

Remarks:

- The new machine FA_3 constructed by the above algorithm will simultaneously keep track of where the input would be if it were running on FA_1 alone, and where the input would be if it were running on FA_2 alone.
- If a string traces through the new machine FA_3 and ends up at a final state, it means that it would also end at a final state either on machine FA_1 or on machine FA_2. Also, any string accepted by either FA_1 or FA_2 will be accepted by this FA_3. So, the language FA_3 accepts is the **union** of the languages accepted by FA_1 and FA_2, respectively.
- **Note** that since there are only finitely many states *x's* and finitely many states *y's*, there can be only finitely many possible states *z's*.
- Let us look at an example illustrating how the algorithm works.

Example:

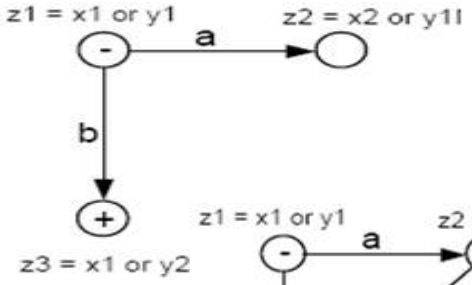
- Consider the following two FAs:



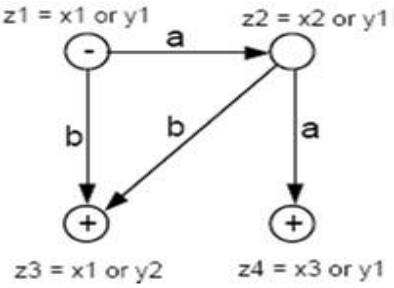
- FA_1 accepts all words with a double a in them.
- FA_2 accepts all words ending with b.
- Let's follow the algorithm to build FA_3 that accepts the union of the two languages.

Combining the FAs:

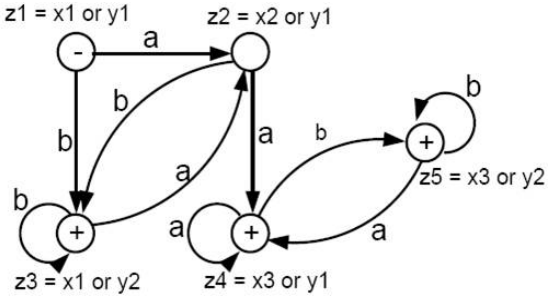
- The start (-) state of FA_3 is $z_1 = x_1$ or y_1.
- In z_1, if we read an *a*, we go to x_2 (observing FA_1), or we go to y_1 (observing FA_2).
  Let $z_2 = x_2$ or y_1.
- In z_1, if we read a *b*, we go to x_1 (observing FA_1), or to y_2 (observing FA_2).
  Let $z_3 = x_1$ or y_2. Note that z_3 must be a final state since y_2 is a final state.



- In z_2, if we read an *a*, we go to x_3 or y_1.
  Let $z_4 = x_3$ or y_1. z_4 is a final state because x_3 is.



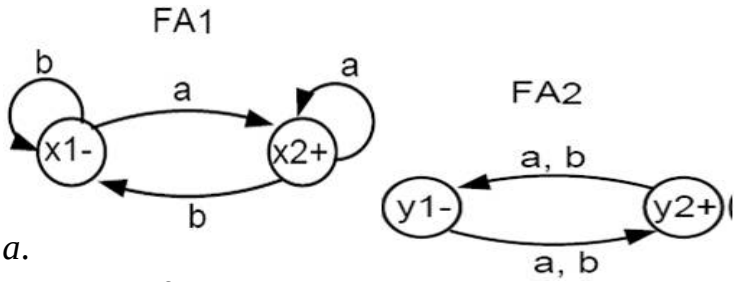
- In z_3, if we read an *a*, we go to x_2 or y_1, which is z_2.
- In z_3, if we read a *b*, we go to x_1 or y_2, which is z_3.
- In z_4, if we read an *a*, we go to x_3 or y_1, which is z_4.
  Hence, we have an a-loop at z_4.
- In z_4, if we read a *b*, we go to x_3 or y_2. Let $z_5 = x_3$ or y_2. Note that z_5 is a final state because x_3 (and y_2) are.
- In z_5, if we read an *a*, we go to x_3 or y_1, which is z_4.
- In z_5, if we read a *b*, we go to x_3 or y_2, which is z_5. Hence, there is a b-loop at z_5.
- The whole machine looks like the following:



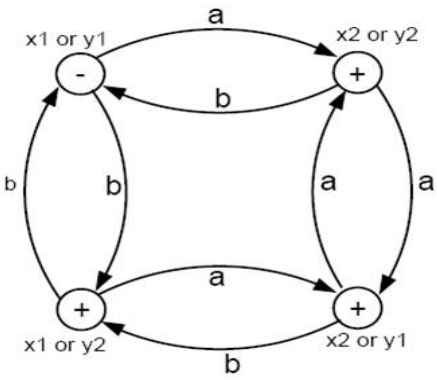
- This machine accepts all words that have a double *a* or that end with *b*.
- The labels $z_1 = x_1$ or y_1, $z_2 = x_2$ or y_1, etc. can be removed if you want.

Example:

- Consider the following two FAs:



- FA_1 accepts all words that end in *a*.
- FA_2 accepts all words with an odd number of letters (odd length).
- Can you use the algorithm to build a machine FA3 that accepts all words that either have an odd number of letters or end in *a*?
- Using the algorithm, we can produce FA_3 that accepts all words that either have an odd number of letters or end in a, as follows:



- The only state that is not a + state is the - state. To get back to that start state, a word must have an even number of letters **and** end in *b*.

Rule 3:

If there is an FA_1 that accepts the language defined by the regular expression r_1, and there is an FA_2 that accepts the language defined by the regular expression r_2, then there is an FA_3 that accepts the language defined by the (concatenation) regular expression (r_1r_2), i.e. the product language.

- We shall show that such an FA_3 exists by presenting an algorithm showing how to construct it from FA_1 and FA_2.
- The idea is to construct a machine that starts out like FA_1 and follows along it until it enters a final state at which time an option is reached. Either we continue along FA_1, waiting to reach another +, or else we switch over to the start state of FA_2 and begin circulating there.